The Report committee for Sarvesh Velore Nagarajan

Certifies that this is the approved version of the following report:

Automated Test Input Generation and Test Execution for Websites

APPROVED BY

SUPERVISING COMMITTEE:

Supervisor: _____

Sarfraz Khurshid

_____

K. Suzanne Barber

Automated Test Input Generation and Test Execution for Websites

by

Sarvesh Velore Nagarajan, B.S.E.E.

Report

Presented to the Faculty of the Graduate School

of the University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2015

Acknowledgements

I extend my sincere gratitude to my supervisor Dr. Sarfraz Khurshid, without whose guidance the completion of this report would not have been possible. The concepts of software testing and validation that I learnt in Dr. Khurshid's class were the inspiration for this report.

I express my sincere gratitude to Dr. Suzanne Barber for accepting to be the reader for my Masters Report. Dr. Barber's advice and feedback have contributed immensely to the successful completion of this report.

I also thank Dr. Nastaran Shafiei for helping review my work and provide valuable feedback for my work in this report. She is also the creator of jpf-nhandler, a JPF extension without which the implementation of the work in this report would not have been possible.

Automated Test Input Generation and Test Execution for Websites

by

Sarvesh Velore Nagarajan, M.S.E.

The University of Texas at Austin, 2015

SUPERVISOR: Sarfraz Khurshid

This report presents a framework for automating test input generation, test execution and validation of websites. The framework leverages popular libraries available for the Java language to perform functions such as interacting with a website, parsing the content of a webpage and writing unit tests. Formal methods for software verification are incorporated using Java Pathfinder (JPF), a model checker built using Java. A working prototype of this framework has been created and tested on a sample website built as part of the work presented in this report. The framework presented in this report can provide full round-trip coverage of user interactions with a website validating website front-end as well as backend execution paths. How this testing framework can be run as part of a continuous integration solution is also presented in this report and has been prototyped. The framework presented in this report can be applied to a variety of real-world websites that involve user interaction.

Contents

List of Figures

List of Tables

1 Introduction

Websites are ubiquitous in today's world. People are constantly interacting with websites from their computers or mobile devices. According to [1], 72% of people surveyed said they ordered pizza online. Furthermore, there is a rise in the number of people using mobile apps to search for restaurants, get restaurant reviews, order food, etc. According to [2], digital banking is on the rise as is evident with the use of online and mobile banking. What all this means is that there is plenty of user interaction between users and websites. Users send information via a user interface which necessitates a response from the websites and perhaps includes modifying state in the website backend. Given such a technological environment there is a need for more comprehensive testing of the website user interface as well as validation of the effects of the interaction in an exhaustive manner. This report presents a framework to test websites through exhaustive test input generation as well as automated execution of the test inputs on the websites and validation of website backend state.

2 Motivation

So far we have mentioned how websites and web applications have become widespread and have given examples of domains where their use has been on the rise. In particular, we see how user interaction with websites has increased for everything from ordering food to performing banking transactions. It is imperative therefore, to come up with more exhaustive testing of websites. So the question becomes, what software out there can we use to help build a website testing framework? The adoption of the Java programming language has been increasing and with its rise, there has been an increase in the number of Java libraries that cater to a wide variety of domains and applications. Selenium [3] is a web browser automation tool for Java. It can simulate user interaction with a website by allowing one to programmatically interact with a website from Java. Junit [4] is a unit testing framework for Java. It allows for writing effective Java unit tests with effective assertion checking. JSoup [5] is a HTML DOM parser. It can help scrape a website's HTML content for interesting pieces such as form fields. Java Pathfinder (JPF) [6] is a model checker for Java. JPF supports state space exploration using backtracking and so caters well to test input generation applications. In recent times there has been increased interest in being able to apply model checking tools such as JPF to help validate real-world applications. The authors of [7] present a tool built on JPF that generates a high coverage test suite for Industrial Java applications which support strings. The authors of [8] present techniques for testing distributed systems written in Java using JPF. In [9], the authors present ways to find bugs in computer games written in Java using JPF. Given the maturity and power of these technologies, the time is ripe to develop a website testing framework. The following sections of this report describe the design of this framework. In this report we concern ourselves

primarily with test input generation for HTML forms, applying these inputs to websites in an automated fashion and validating the results. Websites that contain sequences of webpages that accept form input include websites such as those accepting pizza orders, booking a rental car or an airline ticket.

3 High Level Design

The design of the website testing framework is organized into three main phases. The output of each phase is the input of the subsequent phase. JSON is the format chosen for the output of each phase. The reason being that JSON is a widely used format, there are many libraries and tools to create, format and parse JSON strings. The use of the JSON format, therefore allows for a modular and scalable design where we can have the implementations of the different phases swapped out with alternate implementations and still be able to accept the outputs of the previous phase in a well understood format.

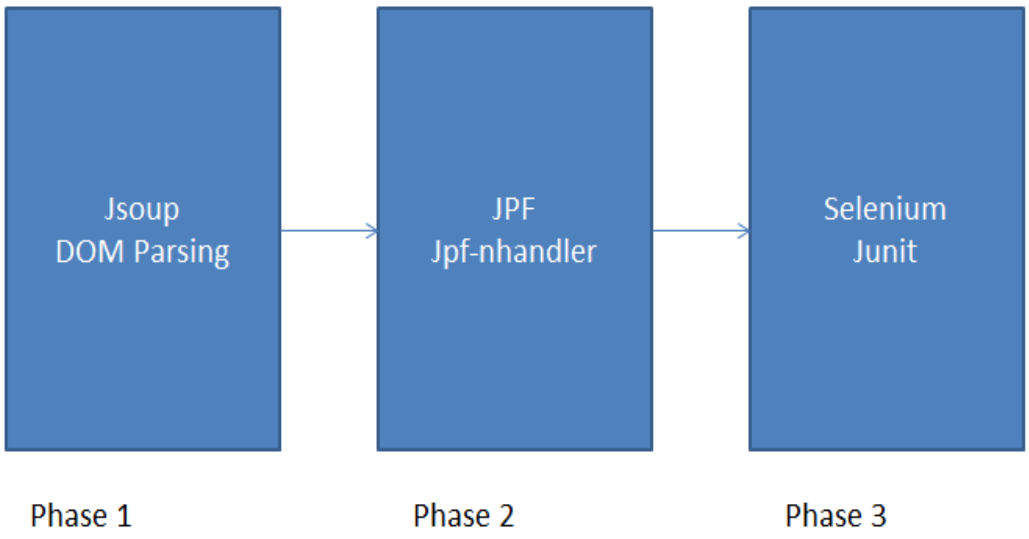

Figure 3-1. The three phases that make up the website testing framework

The first phase involves parsing the DOM of the webpage and obtaining form information. The form elements along with drop down menu options (if any) and string input options are formatted into JSON and sent to the second phase.

In the second phase we use the JPF model checker to generate all possible input combinations for the webpage form. These input combinations form the test cases which are then sent to the third phase which is the Selenium phase. The test inputs generated in this phase are output in JSON. In order to use third-party libraries such as GSON, as well as Java features such as the Files API inside the JPF environment, we use jpf-nhandler. Jpf-nhandler directs calls to the native JVM rather than the JPF JVM and so allows for the use of more complex libraries and language features inside the JPF environment.

In the third phase we use the Selenium web automation tool to exercise the website using the test inputs obtained from phase 2. The Selenium code is executed within the junit unit testing environment. Once the form is submitted to the website, the backend sanctity of the website can be tested by sending an HTTP request to the website backend. This HTTP request is wrapped by a function call called RepOK which has the following signature

boolean RepOK ()

A RepOk method checks whether any invariants or constraints have been violated for a particular entity such as a data structure. Korat [10] is a framework for generating test inputs making use of a RepOk method to validate generated test cases.

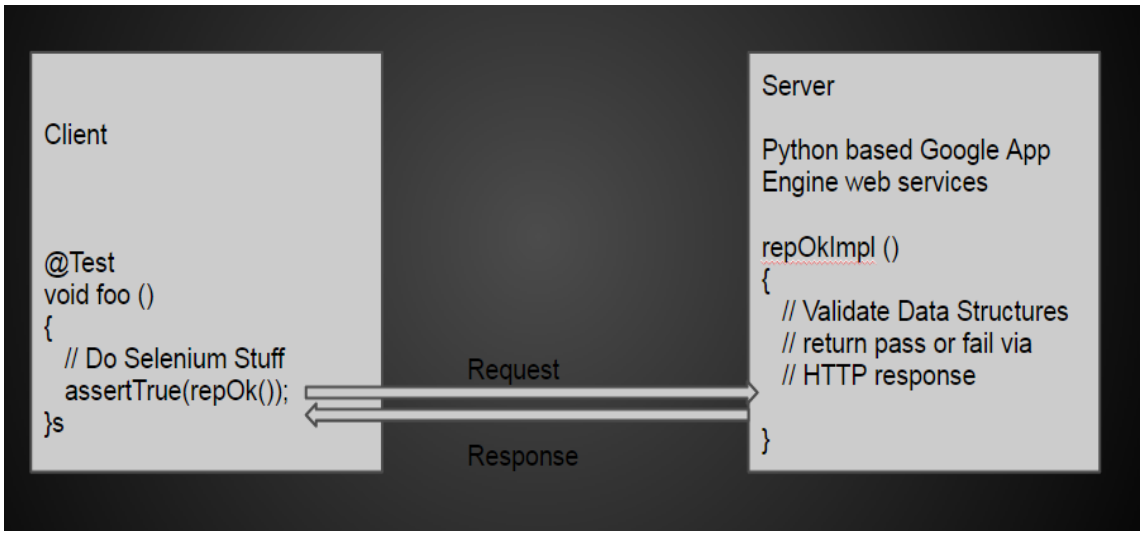

Figure 3-2. Shows the flow of execution when repOk is called in junit on the client side. A HTTP request is sent to the server to execute a repOk function on the server side.

The above figure shows how junit can check an assertion by calling the repOk method on the client side. The repOk method sends an HTTP request to the server. The implementation of the repOk method on the server side will validate the sanctity of any data structures that we would like to validate as part of the test run.

So far we have seen how we can test a form on a webpage by running it through the three phases described above. The DOM of the webpage is parsed to get the form fields, JPF is used to do test input generation, Selenium is used to execute the test inputs on the website and junit is used to validate that the test did not violate any invariants and ensure the sanctity of backend data structures. However, consider a website where subsequent pages resulting from submitting forms end up operating on the same data structures repeatedly. This brings us to the

concept of designing our website for testability. Designing systems for testability is a common design goal. Software architectures should lend themselves to being adequately testable. In the case of websites, since we want to exhaustively test what happens to backend data structures when forms that manipulate them are put through exhaustive test inputs from subsequent pages, we encode a pointer to the data in the URL of the webpage. Then, to exhaustively test a sequence of webpages that operate on the same backend data, we cycle through the same three phases as many times as the number of web pages we want to walk through submitting a form on each page.

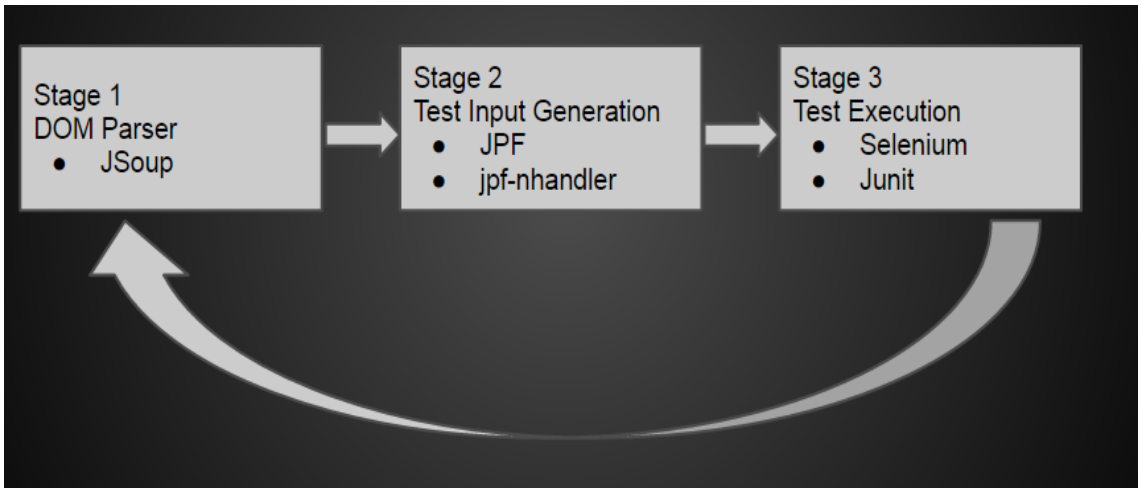

Figure 3-3. Shows the flow of information through the different phases with the urls obtained as a resulting of executing phase 3 being passed back to phase 1.

When going through a sequence of web pages, when Selenium lands on a new page following submitting a form, it logs it and that website is sent to phase 1 for parsing and the three phase process begins for the new web page.

Here is an example of a URL with the pointer to the data encoded in it

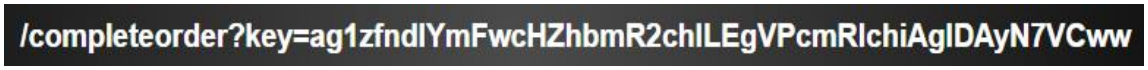

Figure 3-4. Shows the encoding of the datastore key of a website data structure into the URL to aid with exhaustive testability of objects across multiple iterations of the three phase process.

There is a need to be able to run these tests periodically and have a standard way to return results. This can be achieved using Jenkins [11], an open-source continuous integration framework.

4 Detailed Design and Implementation

This section of the report dives into the details of the design and the implementation of some of the key pieces of the testing framework. The website design is described first followed by the Selenium, JPF and JSoup phase respectively.

4.1 Website Design

This section describes the design of the sample website was created for testing the framework described in this report.

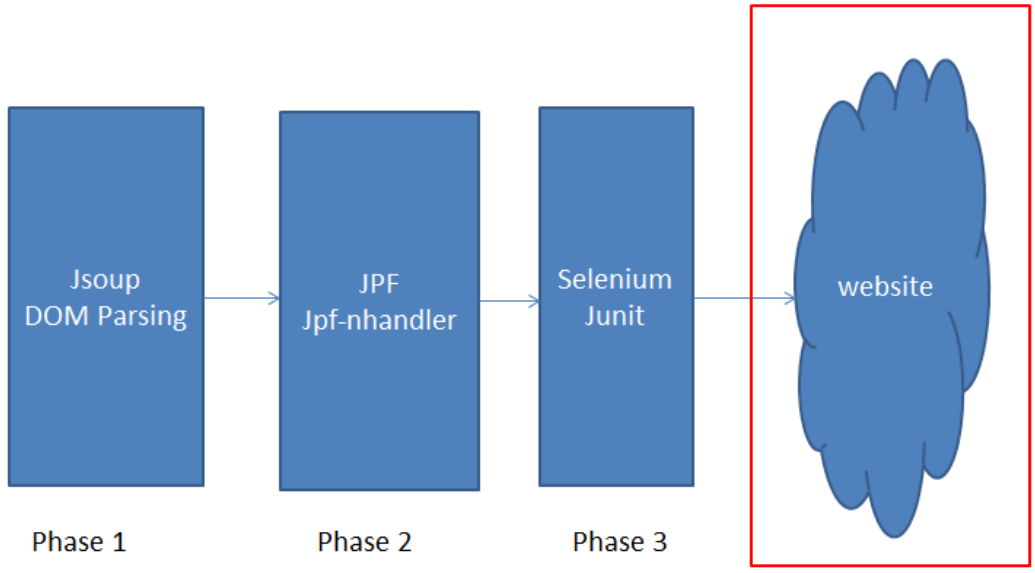

Figure 4-1. Shows the Selenium phase interacting with the sample website created using Google App Engine.

In order to create a working prototype of the design detailed in the High-Level design section, a simple website was created using Google App Engine [12]. The website accepts orders for sandwiches and consists of two webpages. The first page allows a user to pick the type of sandwich and the second page allows them to confirm their order by selecting a payment type.

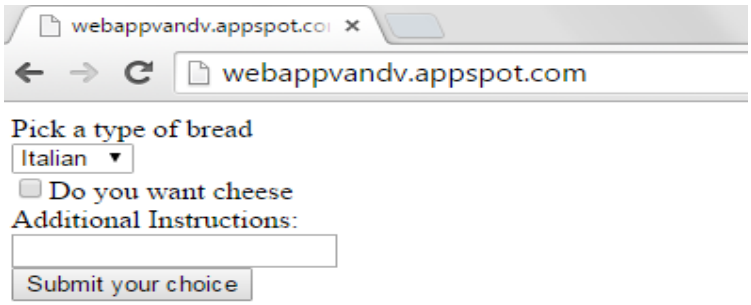

Figure 4-2. Shows the first page of the sample website created using Google App Engine in order to test the website testing framework.

The above figure shows the first page of the website. There is a form that has three fields, each of which is a different type. The type of bread is a dropdown, the cheese selection is a checkbox and the additional instructions can be input as a string.

```
<form name="myform" action="/" method="POST">
            Pick a type of bread <br>
            <select name="mydropdown">
            <option value="Italian">Italian</option>
            <option value="Wheat">Wheat</option>
            <option value="White">White</option>
            </select><br>
            <input type="checkbox" name="cheese" value="yes">Do you want cheese<br>
            Additional Instructions:<br>
            <input type="text" name="additionalInstructions">
            <br>
            <input type="submit" value="Submit your choice">
            </form>
```

Figure 4-3. HTML content of the first page of the sample website. It contains a form that allows a user to select the type of sandwich they want.

The above figure shows the html source for the first page of the website.

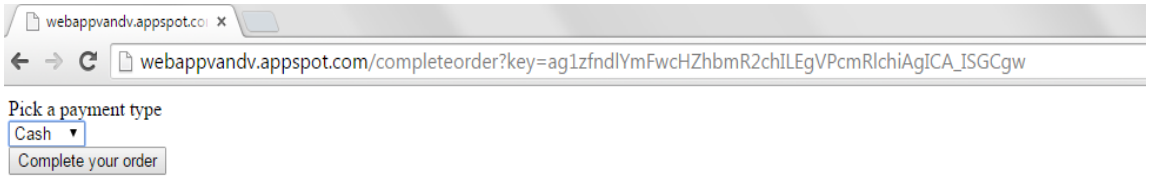

Figure 4-4. Shows the first page of the sample website created using Google App Engine in order to test the website testing framework.

The above figure shows the second page of the website which allows a user to confirm the order by selecting a payment type. The payment type field is a dropdown menu. Note that the URL contains a key which is a pointer to a backend Datastore entry that represents the current order.

```
<form name="myform" action="completeorder" method="POST">
    Pick a payment type <br>
    <select name="paymentType">
    <option value="Card">Card</option>
    <option value="Cash">Cash</option>
    <option value="Check">Check</option>
    </select><br>
     <input type="hidden" name="key" value="ag1zfndlYmFwcHZhbmR2chILEgVPcmRlchiAgICAuv-FCgw">
    <input type="submit" value="Complete your order">
</form>
```

Figure 4-5. HTML content of the second page of the sample website. It contains a form that allows a user to select the type of payment they want to use for their sandwich.

The above figure shows the HTML source for the second page of the website.

The website is written in python and the HTML for the website is created using the jinja2 [13] templating library. The storage of persistent information is done using the Google App Engine datastore.

One can communicate with the website using HTTP requests. One such request calls a method to validate all the orders that are in the system and ensure that certain invariants are always valid. This method is called a repOK method and can be invoked at any time to check the sanctity and integrity of the data structures on the server.

```
class RepOkHandler (webapp2.RequestHandler):
    def post(self):
        output = dict()
        q = Order.all()
        allOk = True
        for i in q:
            if i.orderState == 'Complete' and not i.paymentType:
                allOk = False
                break
        if allOk == True:
            output['result'] = 'OK'
        else:
            output['result'] = 'NOT_OK'
        jsonEncoder = json.JSONEncoder()
        outputJson = jsonEncoder.encode(output)
        self.response.headers['Content-Type'] = 'application/json'
        self.response.write(outputJson)
```

Figure 4-6. The repOk method implementation on the server side written in python.

The above figure shows the code for the repOK method. It checks all orders and validates that if the order state is “Complete”, then a valid payment type has been selected. If all orders satisfy this invariant it returns “OK”, otherwise it returns “NOT_OK”.

4.2 Client Design

All three phases of the testing framework run on the client machine with interactions with the website server during the first and third phase which are the JSoup and Selenium phase respectively. The second phase uses the JPF model checker to generate test inputs. This section covers the design and implementation of all the three phases.

4.2.1 Selenium Phase Design

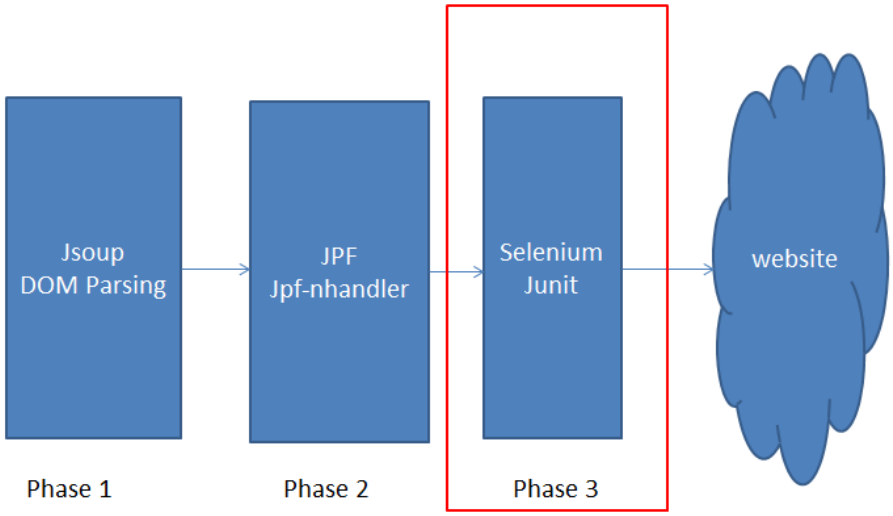

Figure 4-7. Shows the Selenium and Junit phase of the framework which is responsible for executing tests.

On the client side we interact with the server using Selenium. Selenium is a web automation library that allows one to programmatically simulate user interaction with a website. We use Selenium to apply test inputs to our sandwich ordering website. Using Selenium allows for simulating user interaction very accurately thereby providing the ability to test applications the way they will be used in real life.

```
for (FormField field : fields) {
    element = driver.findElement(By.name(field.fieldName));
    switch (field.fieldType) {
        case "checkbox":
            if (field.fieldValue.equals("true")) {
                if (!element.isSelected()) {
                    element.click();
                }
            } else if (field.fieldValue.equals("false")) {
                if (element.isSelected()) {
                    element.click();
                }
            }
            break;
        case "dropdown":
            new Select(element).selectByValue(field.fieldValue);
            break;
        case "text":
            element.sendKeys(field.fieldValue);
        case "submit":
            // We always want to handle the submit button last, but
            // the order in a map is unknown
            break;
    }
```

Figure 4-8. Shows the population of form fields using Selenium. This simulates a user entering input into the form fields from a web browser.

The above code shows the population of form fields using Selenium. The test inputs are executed using Selenium.

The Selenium code is executed from within junit, a Java unit testing framework. Once a test case is executed using Selenium, the sanctity of backend data structures can be validated using the repOk method. The repOk method in the server can be invoked from the client using an HTTP request.

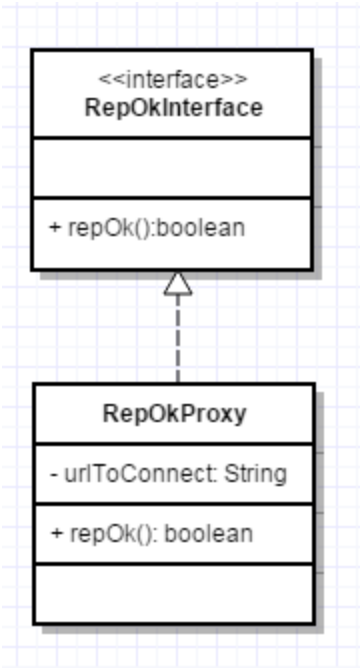

Figure 4-9. Shows the inheritance hierarchy of the repOk classes on the client side. The RepOkProxy class implements the RepOkInterface interface.

For modularity of the design and to cater to greater scalability, we have a RepOkInterface which we define. The junit assertions are written in terms of the interface class and at runtime the appropriate implementation will be called.

```
public interface RepOkInterface {
    public boolean repOk ();
}
```

Figure 4-10. The RepOkInterface definition in Java.

```
public class RepOkProxy implements RepOkInterface{
    private String urlToConnect = null;
    RepOkProxy (String urlToConnect)
    {
        this.urlToConnect = urlToConnect;
    }
```

Figure 4-11. Code showing the RepOkProxy implementing the RepOkInterface.

```
public boolean repOk ()
{
    URL url = null;
    URLConnection connection = null;
    try {
        url = new URL(urlToConnect);
    }
    catch (MalformedURLException e)
    {
        e.printStackTrace();
    }
    try {
        connection = url.openConnection();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
    connection.setDoOutput(true);
    try {
        OutputStreamWriter out = new OutputStreamWriter(
                connection.getOutputStream());
        out.write ("helloworld");
        out.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        BufferedReader in = new BufferedReader(
                            new InputStreamReader(connection.getInputStream())
                            );
        String output = null;
        while ((output = in.readLine()) != null) {
            RepOkResponse response = new Gson().fromJson(output, RepOkResponse.class);
            in.close();
            if (response.result.equalsIgnoreCase("OK"))
                return true;
            else
                return false;
        }
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }

    return false;
}

class RepOkResponse
{
    String result = null;
}
```

Figure 4-12. Implemetation of the repOk method in the RepOkProxy class showing the call to the backend using a HTTP request.

The code in the above figure shows the implementation of the repOk method on the client side. The client sends an HTTP POST request to the server which in turn executes the repOk method on the server. This repOk method can be called following the execution test inputs on the website and junit assertions can be used to validate that repOk succeeds.

```
RepOkInterface ro = new RepOkProxy("http://webappvandv.appspot.com/repok");
```

Figure 4-13. Creating a RepOkProxy instance passing in the URL to send a HTTP request to in order to validate the website backend.

```
Assert.assertEquals(true, ro.repOk());
```

Figure 4-14. Shows a junit assertion that checks that repOk returned true after the execution of test inputs.

4.2.2 JPF Phase Design

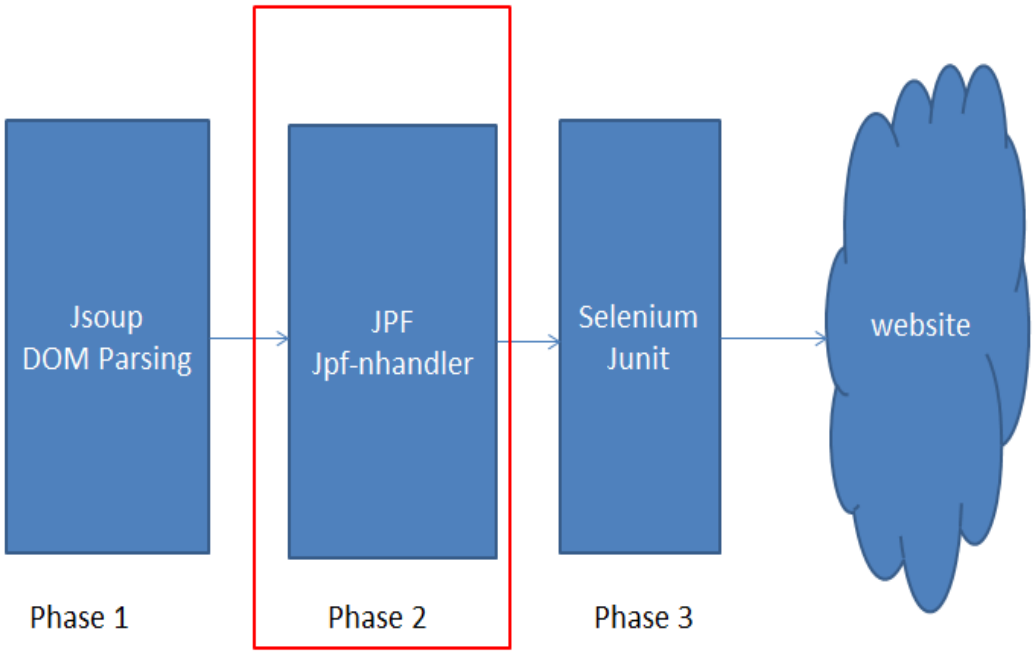

Figure 4-15. The JPF phase uses the JPF model checker to generate test inputs for the Selenium or Junit phase.

The JPF phase is responsible for test input generation. JPF is a model checker that can be used to perform test input generation using non-deterministic choice. Using non-deterministic choice we can generate all permutations of the possible values for the form data for the current page. The test inputs generated at the end of this phase are stored in JSON format.

The non-deterministic choice is made using the JPF Verify API. In particular, the following methods are used to make a non-deterministic choice.

Verify.getBoolean()

Verify.getInt (0, n)

In the current implementation, test input generation is supported for forms that have the following field types:

1. Checkbox
2. Dropdown
3. Strings

The form field types and possible values for the fields are passed to the JPF phase in JSON format whose contents adhere to the layout described by the following Java classes.

```
public static class TestCase
{
        public String type;
        public ArrayList<String> values;
}

public static class TestExplorationMap
{
        public Map<String, TestCase> testExplorationMap;
}
```

Figure 4-16. Shows the data structures used for holding the form fields and possible values for the fields in the JPF phase.

The TestCase class contains two fields; the type of the form field and the possible values for the field. The TestExplorationMap class contains a map whose contents are a set of TestCase objects. The TestExplorationMap therefore represents the different fields and the possible

values for each field for a particular webpage. We can convert a JSON string to an instance of the TestExplorationMap and vice-versa using the GSON [14] library.

```
{"testExplorationMap":{"mydropdown":{"type":"dropdown","values":["Italian","Wheat","White"]},
"cheese":{"type":"checkbox","values":["true","false"]}}}
```

Figure 4-17. A JSON string that can be converted to an instance of TestExplorationMap.

The above figure shows a JSON string that can be converted to an instance of the TestExplorationMap Java class using GSON. This string is an example of input to the JPF phase.

Using libraries such as GSON, in the JPF environment does present a challenge as is better illustrated by the following figure from the JPF website [15].

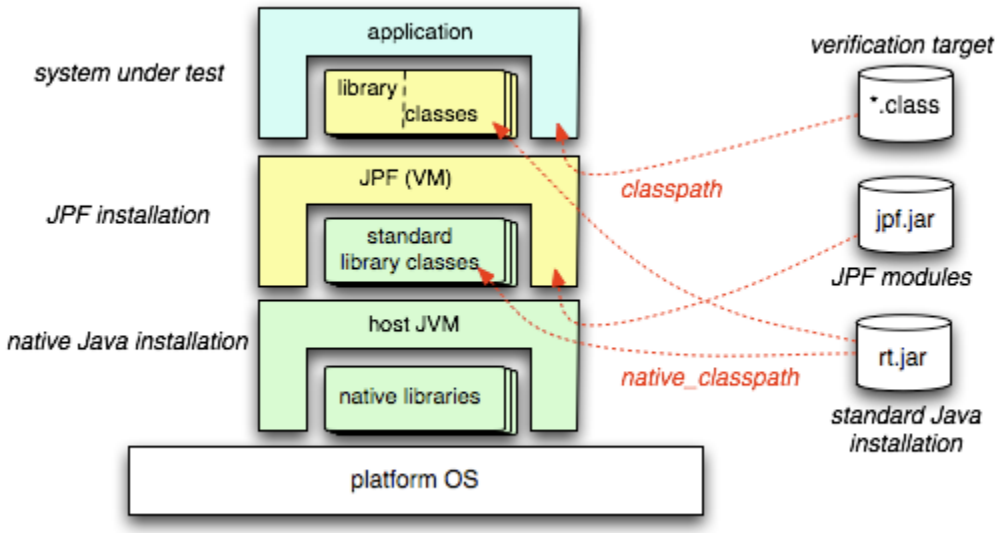

Figure 4-18. Figure showing the layers of the JPF software stack.

When Java code is run on JPF, it actually is executed by the JPF virtual machine. The JPF virtual machine needs the host Java virtual machine in order to execute since it is written in Java.

However, because it is the JPF virtual machine that is executing the code under test, it does not handle complex real-world Java applications. This is the problem we run into with GSON.

In order to tackle this challenge of running complex Java libraries in the JPF environment, we use a library called jpf-nhandler [16]. What jpf-nhandler accomplishes, is it allows for the delegation of function calls in the code under test directly to the host JVM rather than the JPF virtual machine. This allows for the execution of complex Java libraries in the JPF environment allowing for the ease of development of complex applications in the JPF environment.

The following figure shows how to configure JPF to use jpf-nhandler as well as specify which function calls to delegate directly to the host JVM.

```
@using = jpf-nhandler

target = TestJPF

nhandler.resetVMState = false;
nhandler.delegateUnhandledNative = true;
nhandler.spec.delegate =\
 com.google.gson.Gson.*,\
 java.nio.file.Files.readAllBytes,\
 java.nio.file.Paths.get,\
 java.nio.file.Files.write,\
 java.io.IOException.*\

classpath =\
 ./gson-2.3.1.jar;\
 .;\

native_classpath =\
 ./gson-2.3.1.jar;\
 .;\
```

Figure 4-19. Shows the *.jpf file when for the JPF phase when jpf-nhandler is used.

The above lines go into the *.jpf which is used to configure the behavior of JPF. In the file we are specifying that all calls from com.google,gson.Gson needs to be delegated to the host JVM and not be executed by the JPF JVM.

Another challenge that we encounter while writing code in the JPF environment is how to persist state in the face of backtracking. Since JPF is a model-checker, it performs recursive backtracking while performing state space exploration. Any state you have in variables is restored to a previous state when backtracking happens. Therefore, we cannot keep track of all the test inputs generated in a variable in the code itself, because when backtracking happens the state will be reset to some previous state which does not have the current test input that was discovered. This challenge is solved by making use of the file system. Every time a new test input is discovered, we go ahead and write it to a file. This way, the file has the discovered test input even after backtracking has happened. The next time we discover a new test input, we read the file, append the test input to the current content of the file and write it back. We use the Java Files API to perform file access and using jpf-nhandler we can use the Files API in the JPF environment.

We also want to keep track of the number of test cases generated so that we can name the test cases using some ordinal number. This is accomplished using the JPF Verify API which allows for state persistence using counters. The methods that are used are:

- *Verify.incrementCounter(n)*
- *Verify.getCounter(n)*
- *Verify.resetCounter(n)*

```
{
    "testcase0":{
        "mydropdown":{
            "type":"dropdown",
            "value":"Italian"
        },
        "cheese":{
            "type":"checkbox",
            "value":"false"
        }
    },
    "testcase1":{
        "mydropdown":{
            "type":"dropdown",
            "value":"Italian"
        },
        "cheese":{
            "type":"checkbox",
            "value":"true"
        }
    },
    "testcase2":{
        "mydropdown":{
            "type":"dropdown",
            "value":"Wheat"
        },
        "cheese":{
            "type":"checkbox",
            "value":"false"
        }
    },
    "testcase3":{
        "mydropdown":{
            "type":"dropdown",
            "value":"Wheat"
        },
        "cheese":{
            "type":"checkbox",
            "value":"true"
        }
    },
```


Figure 4-20. The output produced by the JPF phase.

The above figure shows a sample output generated by the JPF phase, which is sent over to the Selenium phase.

4.2.3 JSoup Phase Design

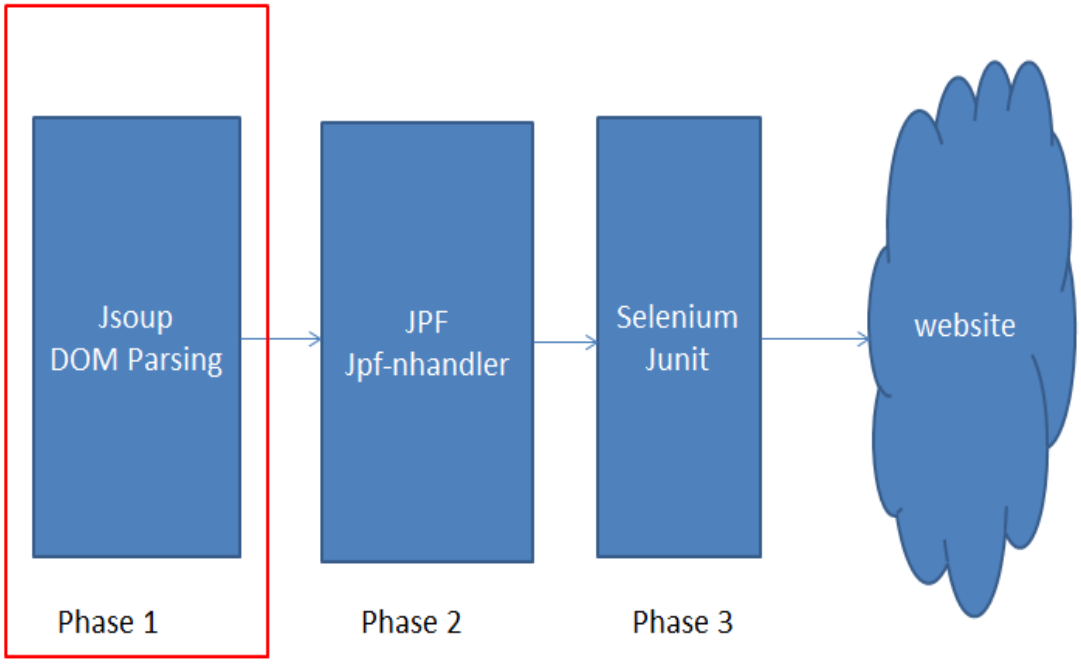

Figure 4-21. The JSoup phase which is responsible for parsing the DOM of the webpage and supplying the possible values of each field of the form on the webpage.

In the JSoup phase, we use the JSoup Java Library to get form fields and possible values for each field. In this phase, JSoup is used to connect to the webpage, scrape the DOM for the field types that are supported, populate the possible values for each field, format this data into JSON and send it to the JPF phase. Again, GSON is used to format the data from an instance of a Java class to a JSON string that can be written to a file and sent over to the JPF phase.

The Java class used to hold the form field and value information must match the class the JPF phase is using to hold the same information. Only, then will we be able to use GSON to format to JSON in the JSoup phase and successfully de-serialize that information in the JPF phase.

```
public static void main(String[] args){
    jsoupParse test = new jsoupParse();
    formFields form = new formFields();
    parseToJson json = new parseToJson();
    BufferedReader br = null;
    try {
        br = new BufferedReader(new FileReader("urls.txt"));
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    String url = null;
    try {
        url = br.readLine();
    } catch (IOException e) {
        e.printStackTrace();
    }
    Document doc = test.parseHTML(url);
    json.parseIntoJson(form.findFields(doc));
```

Figure 4-22. Shows JSoup being used to acquire the DOM content into a Document object.

The above code snippet demonstrates the use of JSoup to connect to a URL of choice specified in a file called urls.txt. The content of the webpage is contained in a Document class which is part of JSoup.

```
try {

        Elements allTypeInput = doc.getElementsByTag("input");
        Elements allDropDown = doc.getElementsByTag("select");

for(Element dropDown:allDropDown){
        Elements option = dropDown.getElementsByTag("option");
        String optionValues = "";
        for (int i=0; i<option.size(); i++){
                optionValues += option.get(i).text() + "\n";
        }
        mapList.add(new SimpleEntry<String,String>(dropDown.attr("name") + "=" + "dropdown", optionValues));
}

for(Element input:allTypeInput){
    if(input.attr("type").equalsIgnoreCase("checkbox")){
        mapList.add(new SimpleEntry<String,String>(input.attr("name") + "=" + "checkbox", "true\nfalse"));
    }
else if(input.attr("type").equalsIgnoreCase("text"))
{
    String optionValues = "";
    for (int i = 0; i < textInputs.size(); i++)
    {
        optionValues+=textInputs.get(i) + "\n";
    }

    mapList.add(new SimpleEntry<String,String>(input.attr("name") + "=" + "text", optionValues));
}
```

Figure 4-23. Shows JSoup used to add form fields and possible values to a map which will later be converted to JSON to be sent to the JPF phase.

The above code shows how to extract form information from a JSoup Document class and adds the form information to a map. The map is later converted to JSON using GSON.

The output of the JSoup phase looks as follows:

```
{
   "testExplorationMap":{
      "mydropdown":{
         "type":"dropdown",
         "values":[
            "Italian",
            "Wheat",
            "White"
         ]
      },
      "cheese":{
         "type":"checkbox",
         "values":[
            "true",
            "false"
         ]
      }
   }
}
```


Figure 4-24. The output of the JSoup phase.

In order to support arbitrary string input into forms, the JSoup phase uses a list of interesting Strings that are part of different input domain spaces within the String input space and adds these strings to the list of possible values for the String form field.

```
no mayo
3 tomatoes
```

Figure 4-25. Shows the contents of a file that contains interesting Strings to try out as part of a test input.

4.2.4 Cycling Through the Phases

The three phases described earlier are responsible for generating and executing test input on the website. The output of phase 1 is fed to phase 2 and the output of phase 2 is fed to phase 3. Phase 1, phase 2 and phase 3 run as separate Java applications reading JSON files for input

parameters. There is therefore a need for a script to compile and run the three phases as well as copy the outputs of one stage into the location where the subsequent stage expects its input.

```
#!/bin/bash
$(cd ../main/resources; cp urls.txt ../java/JsoupPhase)
rm ../main/java/SeleniumPhase/result.txt
for (( i=1; i <= 2; i++ ))
do
    $(cd ../main/java/JsoupPhase; ./runJsoupPhase.sh >/dev/null; cp input.txt ../JpfPhase;
    cp urls.txt ../SeleniumPhase)
    $(cd ../main/java/JpfPhase; ./runExample.sh >/dev/null; cp output.txt ../SeleniumPhase)
    $(cd ../main/java/SeleniumPhase; ./runJunitTest.sh >>result.txt 2>&1; cp urls.txt ../JsoupPhase)
done
cat ../main/java/SeleniumPhase/result.txt
```

Figure 4-26. The shell script used to execute the three phases in a loop.

The above figure shows a shell script that executes each of the phases and copies the output of each phase to the location where the subsequent phase expects the input.

Notice also that that all three phases are executed within a “for” loop. The number of iterations of the “for ” loop controls the number of webpages we want to go through back-to-back. One can see from the last statement within the “for” loop, that a file called urls.txt is copied over to the JSoup phase (phase 1). This file contains the urls of the websites that were landed on while exercising the website through Selenium during the current iteration.

4.2.5 Jenkins Integration

We can have the tests on our website run periodically. This can be handled via a Continuous Integration framework such as Jenkins. We create a Jenkins job for the execution of the tests. The job is configured to call the shell script which cycles through the phases. Using Jenkins

allows us to monitor the pass/fail trend of the tests that we run on the websites as well as ensure that the latest test and website source code are being tested.

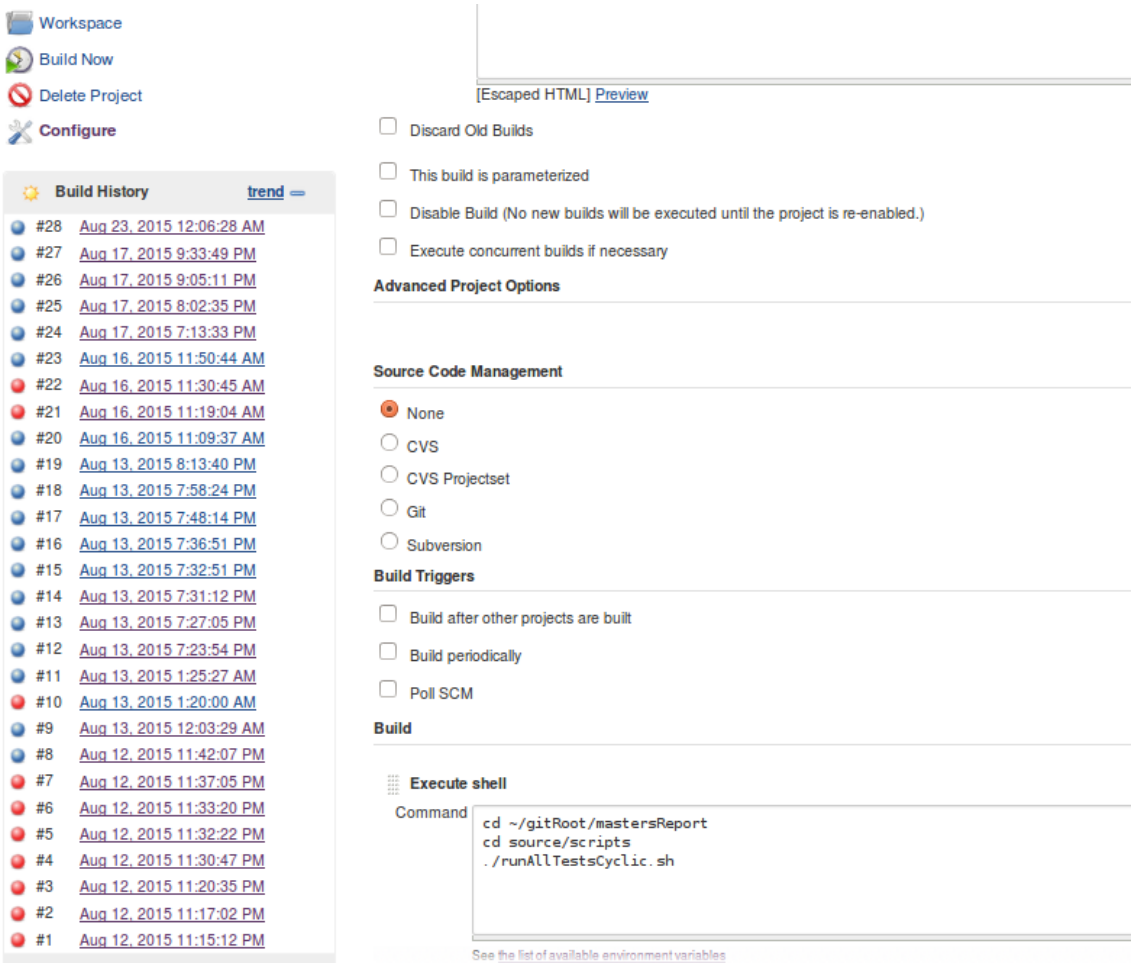

Figure 4-27. The Jenkins job configuration.

5 Results and Observations

The website testing framework described in this report was applied to test the sample website created using Google App Engine. The sample website accepts orders for sandwiches. The testing framework was able to parse the DOM of the pages in the website, extract form information from the DOM, generate all permutations of form input values, submit the forms to the website and validate that the website data structures were left in a valid state. The framework was able to find a bug in the sample website. While developing the sample website there was a bug where when the payment type was being selected, the website read the order information from the datastore for modification but was not storing it back into the datastore. This was discovered by virtue of a failing junit test where the assertion that checked repok to be true failed. The test was successfully executed using a Jenkins job. The framework and its use on a sample website described in this report provides for a mechanism to test other websites that rely on form inputs from the user. Examples include food ordering websites such as pizza ordering sites, hotel booking sites, airline booking sites, etc. The framework demonstrates the use of many widely used Java libraries such as Selenium, Junit, and JSoup as well as libraries that support more formal methods of software validation which in this case is JPF. Furthermore, since source of the test inputs are sourced directly from the DOM of the website, it is data-driven, which is an added advantage as the system can then be self-guiding.

We now discuss the performance of the framework presented in this report. As discussed earlier, the execution of the system flows through three phases and the time taken for the execution of each phase is presented below.

Table 5-1. Shows the time taken for each phase of the framework for two different configurations

Phase	**Time (in seconds)** **Number of form fields = 1** **Total Number of testcases generated = 3**	**Time (in seconds)** **Number of form fields = 3** **Total Number of testcases generated = 12**
Phase 1 (JSoup Phase)	0.426s	1.058s
Phase 2 (JPF Phase)	1.732s	2.233s
Phase 3 (Selenium Phase)	25.23s	104.92s

The above table shows the time it takes to execute each phase of the framework for two different form configurations. The main factors slowing the JSoup phase down would be the network latency and the number of fields in the DOM of the webpage. The network latency hit is a result of the fact that we need to connect to the webpage in order to be able to get the DOM of the webpage. JSoup then creates a Java friendly mechanism to extract information from the DOM and the time it takes to access DOM information using this Java API will be a function of the complexity of the DOM. Therefore the number of fields in the form will play a role in the performance of phase 1. The JPF phase will be slowed down by the fact that exhaustively generating test inputs is an $O(m_1 * m_2 \ldots m_{n-1} * m_n)$ operation. Furthermore, the JPF software stack has more layers than the native Java stack. There is the JPF JVM, jpf-nhandler and other libraries that are used when running in the JPF environment. The Selenium phase performance is impacted once again by network latency. Test cases are executed on the website using a

browser and so a slow network will result in longer test times. The validation of the test results makes use of the repOk method discussed earlier in the report which ends up using an HTTP request to talk to the website. There is also a significant cost associated with launching a web browser such as Mozilla Firefox from Selenium. We also, in the current implementation of the framework, wait for two seconds after the test inputs have been tried on a webpage to find out which webpage we landed on following the submission of the current form.

Table 5-2. Shows the time it takes to test webpages back-to-back

Time (in seconds) for **1 webpage** **12 test cases**	**Time in seconds for** **2 webpages** **12 x 3 = 36 test cases**
107.702s	411.1s

The above table shows the time it takes to test a website consisting of two webpages using the framework presented in this report. It shows the time taken for testing just one page as well as the time it takes to test the first page as well as the one it leads to when tests are executed on the first page. As one can see, the number of test cases rapidly expands in a multiplicative fashion when crawling through more webpages back-to-back.

6 Conclusion

This report presents a framework to automate testing of websites using tools and libraries available for the Java language. The developed framework was used to test a sample website created using Google App Engine. The framework was developed using Selenium, Junit, JSoup and the JPF model checker. Since the framework operates on the website, directly sourcing test input sources from the website and executes generated test input vectors on the website it can scale to real-world production websites. This was proven by creating a simple sandwich ordering website and testing it using the framework. The test framework generates test input vectors exhaustively for forms as well as handles following links that one lands up on form submission. In a world where there are many layers of software from a user's keystrokes to getting a response from a website, the framework presented in this report can help improve software quality by providing exhaustive test coverage.

7 Bibliography

[1] J. Dohnert, "More Consumers Order Food Online Using a Smartphone or Tablet," 28 January 2013. [Online]. Available: http://www.clickz.com/clickz/news/2239608/more-consumers-order-food-online-using-a-smartphone-or-tablet. [Accessed 02 October 2015].

[2] "The irresistible rise of digital banking," [Online]. Available: http://www.bankingtech.com/56242/the-irresistible-rise-of-digital-banking/. [Accessed 1 October 2015].

[3] "What is Selenium?," [Online]. Available: http://www.seleniumhq.org/. [Accessed 1 October 2015].

[4] "JUnit," [Online]. Available: http://junit.org/ . [Accessed 1 October 2015].

[5] "jsoup: Java HTML Parser," [Online]. Available: http://jsoup.org/. [Accessed 1 October 2015].

[6] "JPF ..the swiss army knife of Java verification," [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf. [Accessed 1 October 2015].

[7] I. Ghosh, N. Shafiei, G. Li and W.-F. Chiang, "JST: An automatic test generation tool for industrial Java applications with strings," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[8] N. Shafiei and P. Mehlitz, "Extending JPF to verify distributed systems," *ACM SIGSOFT Software Engineering Notes,* vol. 39, no. 1, pp. 1-5, 2014.

[9] F. van Breugel and N. Shafiei, "Towards model checking of computer games with Java PathFinder," in *Games and Software Engineering (GAS)*, 2013.

[10] C. Boyapati, S. Khurshid and D. Marinov, "Korat: Automated Testing Based on Java Predicates," Cambridge, MA.

[11] "Jenkins," [Online]. Available: https://jenkins-ci.org/. [Accessed 1 October 2015].

[12] "Google App Engine: Platform as a Service," 12 May 2015. [Online]. Available: https://cloud.google.com/appengine/docs. [Accessed 1 October 2015].

[13] "Jinja," [Online]. Available: http://jinja.pocoo.org/. [Accessed 1 October 2015].

[14] I. Singh, J. Leitch and J. Wilson, "Gson User Guide," [Online]. Available: https://sites.google.com/site/gson/gson-user-guide. [Accessed 1 October 2015].

[15] "JPF Components," March 2014. [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf/wiki/user/components. [Accessed 1 October 2015].

[16] N. Shafiei, "jpf-nhandler," [Online]. Available: https://bitbucket.org/nastaran/jpf-nhandler . [Accessed 1 October 2015].